3.5 Dynamic Programming

Dynamic Programming (from now on abbreviated as DP) is perhaps the most challenging problem-solving technique among the four paradigms discussed in this chapter. Thus, make sure that you have mastered the material mentioned in the previous chapters/sections before reading this section. Also, prepare to see lots of recursion and recurrence relations!

The key skills that you have to develop in order to master DP are the abilities to determine the problem *state*s and to determine the relationships or *transitions* between current problems and their sub-problems. We have used these skills earlier in recursive backtracking (see Section 3.2.2). In fact, DP problems with small input size constraints may already be solvable with recursive backtracking.

If you are new to DP technique, you can start by assuming that (the 'top-down') DP is a kind of 'intelligent' or 'faster' recursive backtracking. In this section, we will explain the reasons why DP is often faster than recursive backtracking for problems amenable to it.

DP is primarily used to solve *optimization* problems and *counting* problems. If you encounter a problem that says "minimize this" or "maximize that" or "count the ways to do that", then there is a (high) chance that it is a DP problem. Most DP problems in programming contests only ask for the optimal/total value and not the optimal solution itself, which often makes the problem easier to solve by removing the need to backtrack and produce the solution. However, some harder DP problems also require the optimal solution to be returned in some fashion. We will continually refine our understanding of Dynamic Programming in this section.

3.5.1 DP Illustration

We will illustrate the concept of Dynamic Programming with an example problem: UVa 11450 - Wedding Shopping. The abridged problem statement: Given different options for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, ...) and a certain *limited* budget, our task is to *buy one model of each garment*. We cannot spend more money than the given budget, but we want to spend *the maximum possible* amount.

The input consists of two integers $1 \leq M \leq 200$ and $1 \leq C \leq 20$, where M is the budget and C is the number of garments that you have to buy, followed by some information about the C garments. For the garment `g` $\in$ `[0..`C`-1]`, we will receive an integer $1 \leq K \leq 20$ which indicates the number of different models there are for that garment `g`, followed by K integers indicating the price of each model $\in$ `[1..`K`]` of that garment `g`.

The output is one integer that indicates the maximum amount of money we can spend purchasing one of each garment *without exceeding the budget*. If there is no solution due to the small budget given to us, then simply print "`no solution`".

Suppose we have the following test case A with $M = 20$, $C = 3$:
Price of the 3 models of garment `g = 0` $\rightarrow$ 6 4 $\underline{8}$ // the prices are not sorted in the input
Price of the 2 models of garment `g = 1` $\rightarrow$ 5 $\underline{10}$
Price of the 4 models of garment `g = 2` $\rightarrow$ $\underline{1}$ 5 3 5

For this test case, the answer is 19, which *may* result from buying the underlined items (8+10+1). This is not unique, as solutions (6+10+3) and (4+10+5) are also optimal.

However, suppose we have this test case B with $M =$ **9 (limited budget)**, $C = 3$:
Price of the 3 models of garment `g = 0` $\rightarrow$ 6 4 8
Price of the 2 models of garment `g = 1` $\rightarrow$ 5 10
Price of the 4 models of garment `g = 2` $\rightarrow$ 1 5 3 5

The answer is then "`no solution`" because even if we buy all the cheapest models for each garment, the total price (4+5+1) = 10 still exceeds our given budget $M = 9$.

In order for us to appreciate the usefulness of Dynamic Programming in solving the above-mentioned problem, let's explore how far the *other* approaches discussed earlier will get us in this particular problem.

Approach 1: Greedy (Wrong Answer)

Since we want to maximize the budget spent, one greedy idea (there are other greedy approaches—which are also WA) is to take the most expensive model for each garment `g` which still fits our budget. For example in test case A above, we can choose the most expensive model 3 of garment `g = 0` with price 8 (`money` is now `20-8 = 12`), then choose the most expensive model 2 of garment `g = 1` with price 10 (`money = 12-10 = 2`), and finally for the last garment `g = 2`, we can only choose model 1 with price 1 as the `money` we have left does not allow us to buy the other models with price 3 or 5. This greedy strategy 'works' for test cases A and B above and produce the same optimal solution (8+10+1) = 19 and "`no solution`", respectively. It also runs very fast[8]: $20 + 20 + \ldots + 20$ for a total of 20 times = 400 operations in the worst case. However, this greedy strategy does not work for many other test cases, such as this *counter-example* below (test case C):

Test case C with $M = 12$, $C = 3$:
3 models of garment `g = 0` $\rightarrow$ 6 $\underline{4}$ **8**
2 models of garment `g = 1` $\rightarrow$ $\underline{5}$ 10
4 models of garment `g = 2` $\rightarrow$ 1 5 $\underline{3}$ 5

The Greedy strategy selects model 3 of garment `g = 0` with price **8** (`money = 12-8 = 4`), causing us to not have enough money to buy any model in garment `g = 1`, thus incorrectly reporting "`no solution`". One optimal solution is $\underline{4+5+3}$ = 12, which uses up all of our budget. The optimal solution is not unique as 6+5+1 = 12 also depletes the budget.

Approach 2: Divide and Conquer (Wrong Answer)

This problem is not solvable using the Divide and Conquer paradigm. This is because the sub-problems (explained in the Complete Search sub-section below) are not independent. Therefore, we cannot solve them separately with the Divide and Conquer approach.

Approach 3: Complete Search (Time Limit Exceeded)

Next, let's see if Complete Search (recursive backtracking) can solve this problem. One way to use recursive backtracking in this problem is to write a function `shop(money, g)` with two parameters: The current `money` that we have and the current garment `g` that we are dealing with. The pair `(money, g)` is the *state* of this problem. Note that the order of parameters does not matter, e.g. `(g, money)` is also a perfectly valid state. Later in Section 3.5.3, we will see more discussion on how to select appropriate states for a problem.

We start with `money = M` and garment `g = 0`. Then, we try all possible models in garment `g = 0` (a maximum of 20 models). If model i is chosen, we subtract model i's price from `money`, then repeat the process in a recursive fashion with garment `g = 1` (which can also have up to 20 models), etc. We stop when the model for the last garment `g = C-1` has been chosen. If `money < 0` before we choose a model from garment `g = C-1`, we can prune the infeasible solution. Among all valid combinations, we can then pick the one that results in the smallest non-negative `money`. This maximizes the money spent, which is (`M - money`).

[8]We do not need to sort the prices just to find the model with the maximum price as there are only up to $K \leq 20$ models. An $O(K)$ scan is enough.

We can formally define these Complete Search recurrences (transitions) as follows:
1. If `money < 0` (i.e. money goes negative),
`shop(money, g) =` $-\infty$ (in practice, we can just return a large negative value)
2. If a model from the last garment has been bought, that is, `g = C`,
`shop(money, g) = M - money` (this is the actual money that we spent)
3. In general case, $\forall$ `model` $\in$ `[1..K]` of current garment `g`,
`shop(money, g) = max(shop(money - price[g][model], g + 1))`
We want to maximize this value (Recall that the invalid ones have large negative value)

This solution works correctly, but it is **very slow**! Let's analyze the worst case time complexity. In the largest test case, garment `g = 0` has up to 20 models; garment `g = 1` *also* has up to 20 models and all garments including the last garment `g = 19` ***also*** have up to 20 models. Therefore, this Complete Search runs in $20 \times 20 \times \ldots \times 20$ operations in the worst case, i.e. 20^{20} = a **very large** number. If we can *only* come up with this Complete Search solution, we cannot solve this problem.

Approach 4: Top-Down DP (Accepted)

To solve this problem, we have to use the DP concept as this problem satisfies the two prerequisites for DP to be applicable:

1. This problem has optimal sub-structures[9].
   This is illustrated in the third Complete Search recurrence above: The solution for the sub-problem is part of the solution of the original problem. In other words, if we select model i for garment $g = 0$, for our final selection to be optimal, our choice for garments $g = 1$ and above must also be the optimal choice for a reduced budget of $M - price$, where $price$ refers to the price of model i.
2. This problem has overlapping sub-problems.
   This is the key characteristic of DP! The search space of this problem is *not* as big as the rough 20^{20} bound obtained earlier because **many** sub-problems are *overlapping*!

Let's verify if this problem indeed has overlapping sub-problems. Suppose that there are 2 models in a certain garment `g` with the *same* price `p`. Then, a Complete Search will move to the **same** sub-problem `shop(money - p, g + 1)` after picking *either* model! This situation will also occur if some combination of `money` and chosen model's price causes $\texttt{money}_1$ - $\texttt{p}_1$ = $\texttt{money}_2$ - $\texttt{p}_2$ at the same garment `g`. This will—in a Complete Search solution—cause the same sub-problem to be computed *more than once*, an inefficient state of affairs!

So, how many *distinct* sub-problems (a.k.a. **states** in DP terminology) are there in this problem? Only $201 \times 20 = 4020$. There are only 201 possible values for `money` (0 to 200 inclusive) and 20 possible values for the garment `g` (0 to 19 inclusive). Each sub-problem just needs to be computed *once*. If we can ensure this, we can solve this problem *much faster*.

The implementation of this DP solution is surprisingly simple. If we already have the recursive backtracking solution (see the recurrences—a.k.a. **transitions** in DP terminology—shown in the Complete Search approach above), we can implement the **top-down** DP by adding these two additional steps:

1. Initialize[10] a DP 'memo' table with dummy values that are not used in the problem, e.g. '-1'. The DP table should have dimensions corresponding to the problem states.

[9]Optimal sub-structures are also required for Greedy algorithms to work, but this problem lacks the 'greedy property', making it unsolvable with the Greedy algorithm.

[10]For C/C++ users, the `memset` function in <`cstring`> is a good tool to perform this step.

2. At the start of the recursive function, check if this state has been computed before.
   (a) If it has, simply return the value from the DP memo table, $O(1)$.
   (This the origin of the term 'memoization'.)
   (b) If it has not been computed, perform the computation as per normal (only once) and then store the computed value in the DP memo table so that *further calls* to this sub-problem (state) return immediately.

Analyzing a basic[11] DP solution is easy. If it has M distinct states, then it requires $O(M)$ memory space. If computing one state (the complexity of the DP transition) requires $O(k)$ steps, then the overall time complexity is $O(kM)$. This UVa 11450 problem has $M = 201 \times 20 = 4020$ and $k = 20$ (as we have to iterate through at most 20 models for each garment g). Thus, the time complexity is at most $4020 \times 20 = 80400$ operations per test case, a very manageable calculation.

We display our code below for illustration, especially for those who have never coded a top-down DP algorithm before. Scrutinize this code and verify that it is indeed very similar to the recursive backtracking code that you have seen in Section 3.2.

```
/* UVa 11450 - Wedding Shopping - Top Down */
// assume that the necessary library files have been included
// this code is similar to recursive backtracking code
// parts of the code specific to top-down DP are commented with: 'TOP-DOWN'

int M, C, price[25][25];                 // price[g (<= 20)][model (<= 20)]
int memo[210][25];    // TOP-DOWN: dp table memo[money (<= 200)][g (<= 20)]
int shop(int money, int g) {
  if (money < 0) return -1000000000;     // fail, return a large -ve number
  if (g == C) return M - money;        // we have bought last garment, done
  // if the line below is commented, top-down DP will become backtracking!!
  if (memo[money][g] != -1) return memo[money][g]; // TOP-DOWN: memoization
  int ans = -1;   // start with a -ve number as all prices are non negative
  for (int model = 1; model <= price[g][0]; model++)      // try all models
    ans = max(ans, shop(money - price[g][model], g + 1));
  return memo[money][g] = ans; }     // TOP-DOWN: memoize ans and return it

int main() {            // easy to code if you are already familiar with it
  int i, j, TC, score;
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);                 // store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }
    memset(memo, -1, sizeof memo);    // TOP-DOWN: initialize DP memo table
    score = shop(M, 0);                            // start the top-down DP
    if (score < 0) printf("no solution\n");
    else           printf("%d\n", score);
} } // return 0;
```

[11]Basic means "without fancy optimizations that we will see later in this section and in Section 8.3".

We want to take this opportunity to illustrate another style used in implementing DP solutions (only applicable for C/C++ users). Instead of frequently addressing a certain cell in the memo table, we can use a local *reference* variable to store the memory address of the required cell in the memo table as shown below. The two coding styles are not very different, and it is up to you to decide which style you prefer.

```
int shop(int money, int g) {
  if (money < 0) return -1000000000; // order of >1 base cases is important
  if (g == C) return M - money;  // money can't be <0 if we reach this line
  int &ans = memo[money][g];                  // remember the memory address
  if (ans != -1) return ans;
  for (int model = 1; model <= price[g][0]; model++)
    ans = max(ans, shop(money - price[g][model], g + 1));
  return ans;                // ans (or memo[money][g]) is directly updated
}
```

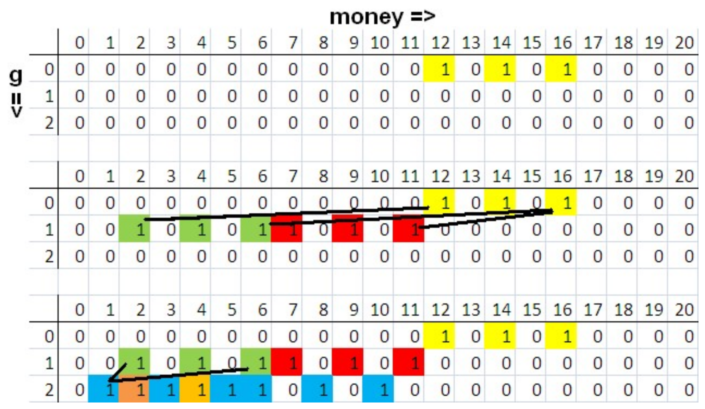
Source code: `ch3_02_UVa11450_td.cpp/java`

Approach 5: Bottom-Up DP (Accepted)

There is another way to implement a DP solution often referred to as the **bottom-up** DP. This is actually the 'true form' of DP as DP was originally known as the 'tabular method' (computation technique involving a table). The *basic* steps to build bottom-up DP solution are as follows:

1. Determine the required set of parameters that uniquely describe the problem (the state). This step is similar to what we have discussed in recursive backtracking and top-down DP earlier.
2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state. This is equivalent to the memo table in top-down DP. However, there are differences. In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases). Recall that in top-down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values.
3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions). Repeat this process until the DP table is complete. For the bottom-up DP, this part is usually accomplished through iterations, using loops (more details about this later).

For UVa 11450, we can write the bottom-up DP as follow: We describe the state of a sub-problem with two parameters: The current garment `g` and the current `money`. This state formulation is essentially equivalent to the state in the top-down DP above, except that we have reversed the order to make `g` the first parameter (thus the values of `g` are the row indices of the DP table so that we can take advantage of cache-friendly row-major traversal in a 2D array, see the speed-up tips in Section 3.2.3). Then, we initialize a 2D table (boolean matrix) `reachable[g][money]` of size 20×201. Initially, only cells/states reachable by buying any of the models of the first garment `g = 0` are set to true (in the first row). Let's use test case A above as example. In Figure 3.8, top, the only columns '20-6 = 14', '20-4 = 16', and '20-8 = 12' in row 0 are initially set to true.



g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure 3.8: Bottom-Up DP (columns 21 to 200 are not shown)

Now, we loop from the second garment `g = 1` (second row) to the last garment `g = C-1 = 3-1 = 2` (third and last row) in row-major order (row by row). If `reachable[g-1][money]` is true, then the next state `reachable[g][money-p]` where `p` is the price of a model of current garment `g` is also reachable as long as the second parameter (`money`) is not negative. See Figure 3.8, middle, where `reachable[0][16]` propagates to `reachable[1][16-5]` and `reachable[1][16-10]` when the model with price 5 and 10 in garment `g = 1` is bought, respectively; `reachable[0][12]` propagates to `reachable[1][12-10]` when the model with price 10 in garment `g = 1` is bought, etc. We repeat this table filling process row by row until we are done with the last row[12].

Finally, the answer can be found in the last row when `g = C-1`. Find the state in that row that is both nearest to index 0 and reachable. In Figure 3.8, bottom, the cell `reachable[2][1]` provides the answer. This means that we can reach state (`money = 1`) by buying some combination of the various garment models. The required final answer is actually `M - money`, or in this case, `20-1 = 19`. The answer is "`no solution`" if there is no state in the last row that is reachable (where `reachable[C-1][money]` is set to true). We provide our implementation below for comparison with the top-down version.

```
/* UVa 11450 - Wedding Shopping - Bottom Up */
// assume that the necessary library files have been included

int main() {
  int g, money, k, TC, M, C;
  int price[25][25];                          // price[g (<= 20)][model (<= 20)]
  bool reachable[25][210];    // reachable table[g (<= 20)][money (<= 200)]

  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);                // we store K in price[g][0]
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
```

[12]Later in Section 4.7.1, we will discuss DP as a traversal of an (implicit) DAG. To avoid unnecessary 'backtracking' along this DAG, we have to visit the vertices in their topological order (see Section 4.2.5). The order in which we fill the DP table is a topological ordering of the underlying implicit DAG.

```
    memset(reachable, false, sizeof reachable);         // clear everything
    for (g = 1; g <= price[0][0]; g++)        // initial values (base cases)
      if (M - price[0][g] >= 0)      // to prevent array index out of bound
        reachable[0][M - price[0][g]] = true;  // using first garment g = 0

    for (g = 1; g < C; g++)                    // for each remaining garment
      for (money = 0; money < M; money++) if (reachable[g-1][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[g][money - price[g][k]] = true;   // also reachable now

    for (money = 0; money <= M && !reachable[C - 1][money]; money++);

    if (money == M + 1) printf("no solution\n");  // last row has no on bit
    else                printf("%d\n", M - money);
  }
} // return 0;
```

Source code: `ch3_03_UVa11450_bu.cpp/java`

There is an advantage for writing DP solutions in the bottom-up fashion. For problems where we only need the last row of the DP table (or, more generally, the last updated slice of all the states) to determine the solution—including this problem, we can optimize the *memory usage* of our DP solution by sacrificing one dimension in our DP table. For harder DP problems with tight memory requirements, this 'space saving trick' may prove to be useful, though the overall time complexity does not change.

Let's take a look again at Figure 3.8. We only need to store two rows, the current row we are processing and the previous row we have processed. To compute row 1, we only need to know the columns in row 0 that are set to true in `reachable`. To compute row 2, we similarly only need to know the columns in row 1 that are set to true in `reachable`. In general, to compute row g, we only need values from the previous row $g - 1$. So, instead of storing a boolean matrix `reachable[g][money]` of size $\underline{20} \times 201$, we can simply store `reachable[2][money]` of size $\underline{2} \times 201$. We can use this programming trick to reference one row as the 'previous' row and another row as the 'current' row (e.g. `prev = 0, cur = 1`) and then swap them (e.g. now `prev = 1, cur = 0`) as we compute the bottom-up DP row by row. Note that for this problem, the memory savings are not significant. For harder DP problems, for example where there might be thousands of garment models instead of 20, this space saving trick can be important.

Top-Down versus Bottom-Up DP

Although both styles use 'tables', the way the bottom-up DP table is filled is different to that of the top-down DP *memo* table. In the top-down DP, the memo table entries are filled 'as needed' through the recursion itself. In the bottom-up DP, we used a correct 'DP table filling order' to compute the values such that the previous values needed to process the current cell have already been obtained. This table filling order is the topological order of the implicit DAG (this will be explained in more detail in Section 4.7.1) in the recurrence structure. For most DP problems, a topological order can be achieved simply with the proper sequencing of some (nested) loops.

For most DP problems, these two styles are equally good and the decision to use a particular DP style is a matter of preference. However, for harder DP problems, one of the styles can be better than the other. To help you understand which style that you should use when presented with a DP problem, please study the trade-offs between top-down and bottom-up DPs listed in Table 3.2.

Top-Down	**Bottom-Up**
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.2: DP Decision Table

Displaying the Optimal Solution

Many DP problems request only for the value of the optimal solution (like the UVa 11450 above). However, many contestants are caught off-guard when they are also required to print the optimal solution. We are aware of two ways to do this.

The first way is mainly used in the bottom-up DP approach (which is still applicable for top-down DPs) where we store the predecessor information at each state. If there are more than one optimal predecessors and we have to output all optimal solutions, we can store those predecessors in a list. Once we have the optimal final state, we can do backtracking from the optimal final state and follow the optimal transition(s) recorded at each state until we reach one of the base cases. If the problem asks for all optimal solutions, this backtracking routine will print them all. However, most problem authors usually set additional output criteria so that the selected optimal solution is unique (for easier judging).

Example: See Figure 3.8, bottom. The optimal final state is `reachable[2][1]`. The predecessor of this optimal final state is state `reachable[1][2]`. We now backtrack to `reachable[1][2]`. Next, see Figure 3.8, middle. The predecessor of state `reachable[1][2]` is state `reachable[0][12]`. We then backtrack to `reachable[0][12]`. As this is already one of the initial base states (at the first row), we know that an optimal solution is: (20→12) = price 8, then (12→2) = price 10, then (2→1) = price 1. However, as mentioned earlier in the problem description, this problem may have several other optimal solutions, e.g. We can also follow the path: `reachable[2][1]` → `reachable[1][6]` → `reachable[0][16]` which represents another optimal solution: (20→16) = price 4, then (16→6) = price 10, then (6→1) = price 5.

The second way is applicable mainly to the top-down DP approach where we utilize the strength of recursion and memoization to do the same job. Using the top-down DP code shown in Approach 4 above, we will add another function `void print_shop(int money, int g)` that has the same structure as `int shop(int money, int g)` except that it uses the values stored in the memo table to reconstruct the solution. A sample implementation (that only prints out one optimal solution) is shown below:

```
void print_shop(int money, int g) {          // this function returns void
  if (money < 0 || g == C) return;                       // similar base cases
  for (int model = 1; model <= price[g][0]; model++) // which model is opt?
    if (shop(money - price[g][model], g + 1) == memo[money][g]) {
      printf("%d%c", price[g][model], g == C-1 ? '\n' : '-');   // this one
      print_shop(money - price[g][model], g + 1);  // recurse to this state
      break;                                       // do not visit other states
}   }
```

Exercise 3.5.1.1: To verify your understanding of UVa 11450 problem discussed in this section, determine what is the output for test case D below?

Test case D with $M = 25$, $C = 3$:
Price of the 3 models of garment `g = 0` $\rightarrow$ 6 4 8
Price of the 2 models of garment `g = 1` $\rightarrow$ 10 6
Price of the 4 models of garment `g = 2` $\rightarrow$ 7 3 1 5

Exercise 3.5.1.2: Is the following state formulation `shop(g, model)`, where `g` represents the current garment and `model` represents the current model, appropriate and exhaustive for UVa 11450 problem?

Exercise 3.5.1.3: Add the space saving trick to the bottom-up DP code in Approach 5!

3.5.2 Classical Examples

The problem UVa 11450 - Wedding Shopping above is a (relatively easy) non-classical DP problem, where we had to come up with the correct DP states and transitions *by ourself*. However, there are many other *classical* problems with efficient DP solutions, i.e. their DP states and transitions are *well-known*. Therefore, such classical DP problems and their solutions should be mastered by every contestant who wishes to do well in ICPC or IOI! In this section, we list down six classical DP problems and their solutions. Note: Once you understand the basic form of these DP solutions, try solving the programming exercises that enumerate their *variants*.

1. Max 1D Range Sum

Abridged problem statement of UVa 507 - Jill Rides Again: Given an integer array `A` containing $n \leq 20K$ non-zero integers, determine the maximum (1D) range sum of `A`. In other words, find the maximum Range Sum Query (RSQ) between two indices `i` and `j` in `[0..n-1]`, that is: `A[i] + A[i+1] + A[i+2] +...+ A[j]` (also see Section 2.4.3 and 2.4.4).

A Complete Search algorithm that tries all possible $O(n^2)$ pairs of `i` and `j`, computes the required `RSQ(i, j)` in $O(n)$, and finally picks the maximum one runs in an overall time complexity of $O(n^3)$. With n up to $20K$, this is a TLE solution.

In Section 2.4.4, we have discussed the following DP strategy: Pre-process array `A` by computing `A[i] += A[i-1]` $\forall$`i` $\in$ `[1..n-1]` so that `A[i]` contains the sum of integers in subarray `A[0..i]`. We can now compute `RSQ(i, j)` in $O(1)$: `RSQ(0, j) = A[j]` and `RSQ(i, j) = A[j] - A[i-1]` $\forall$`i > 0`. With this, the Complete Search algorithm above can be made to run in $O(n^2)$. For n up to $20K$, this is still a TLE approach. However, this technique is still useful in other cases (see the usage of this 1D Range Sum in Section 8.4.2).

There is an even better algorithm for this problem. The main part of Jay Kadane's $O(n)$ (can be viewed as a greedy or DP) algorithm to solve this problem is shown below.

```
// inside int main()
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 };   // a sample array A
  int sum = 0, ans = 0;           // important, ans must be initialized to 0
  for (int i = 0; i < n; i++) {                           // linear scan, O(n)
    sum += A[i];                      // we greedily extend this running sum
    ans = max(ans, sum);                  // we keep the maximum RSQ overall
    if (sum < 0) sum = 0;                    // but we reset the running sum
  }                                                // if it ever dips below 0
  printf("Max 1D Range Sum = %d\n", ans);
```

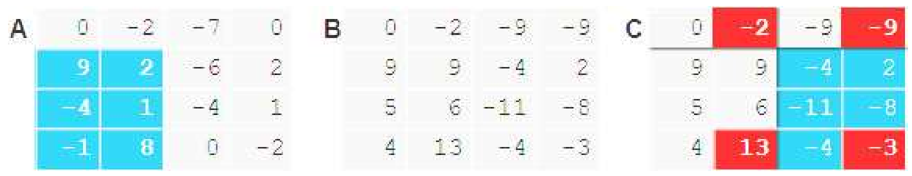
Source code: `ch3_04_Max1DRangeSum.cpp/java`

The key idea of Kadane's algorithm is to keep a running sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0. This is because re-starting from 0 is always better than continuing from a negative running sum. Kadane's algorithm is the required algorithm to solve this UVa 507 problem as $n \leq 20K$.

Note that we can also view this Kadane's algorithm as a DP solution. At each step, we have two choices: We can either leverage the previously accumulated maximum sum, or begin a new range. The DP variable `dp(i)` thus represents the maximum sum of a range of integers that ends with element `A[i]`. Thus, the final answer is the maximum over all the values of `dp(i)` where i $\in$ `[0..n-1]`. If zero-length ranges are allowed, then 0 must also be considered as a possible answer. The implementation above is essentially an efficient version that utilizes the space saving trick discussed earlier.

2. Max 2D Range Sum

Abridged problem statement of UVa 108 - Maximum Sum: Given an $n \times n$ $(1 \leq n \leq 100)$ square matrix of integers `A` where each integer ranges from `[-127..127]`, find a sub-matrix of `A` with the maximum sum. For example: The 4×4 matrix $(n = 4)$ in Table 3.3.A below has a 3×2 sub-matrix on the lower-left with maximum sum of 9 + 2 - 4 + 1 - 1 + 8 = 15.



A				
	0	-2	-7	0
	9	2	-6	2
	-4	1	-4	1
	-1	8	0	-2

B				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

C				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

Table 3.3: UVa 108 - Maximum Sum

Attacking this problem naïvely using a Complete Search as shown below does not work as it runs in $O(n^6)$. For the largest test case with $n = 100$, an $O(n^6)$ algorithm is too slow.

```
maxSubRect = -127*100*100;    // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {    // end coord
    subRect = 0;                         // sum the items in this sub-rectangle
    for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
      subRect += A[a][b];
    maxSubRect = max(maxSubRect, subRect); }          // the answer is here
```

The solution for the Max 1D Range Sum in the previous subsection can be extended to two (or more) dimensions as long as the inclusion-exclusion principle is properly applied. The only difference is that while we dealt with overlapping sub-ranges in Max 1D Range Sum, we will deal with overlapping sub-matrices in Max 2D Range Sum. We can turn the $n \times n$ input matrix into an $n \times n$ *cumulative sum matrix* where `A[i][j]` no longer contains its own value, but the sum of all items within sub-matrix `(0, 0)` to `(i, j)`. This can be done simultaneously while reading the input and still runs in $O(n^2)$. The code shown below turns the input square matrix (see Table 3.3.A) into a cumulative sum matrix (see Table 3.3.B).

```
scanf("%d", &n);                    // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];             // if possible, add from top
  if (j > 0) A[i][j] += A[i][j - 1];            // if possible, add from left
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];     // avoid double count
}                                               // inclusion-exclusion principle
```

With the sum matrix, we can answer the sum of any sub-matrix `(i, j)` to `(k, l)` in $O(1)$ using the code below. For example, let's compute the sum of `(1, 2)` to `(3, 3)`. We split the sum into 4 parts and compute `A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9` as highlighted in Table 3.3.C. With this $O(1)$ DP formulation, the Max 2D Range Sum problem can now be solved in $O(n^4)$. For the largest test case of UVa 108 with $n = 100$, this is still fast enough.

```
maxSubRect = -127*100*100;    // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {    // end coord
    subRect = A[k][l];      // sum of all items from (0, 0) to (k, l): O(1)
    if (i > 0) subRect -= A[i - 1][l];                              // O(1)
    if (j > 0) subRect -= A[k][j - 1];                              // O(1)
    if (i > 0 && j > 0) subRect += A[i - 1][j - 1];                 // O(1)
    maxSubRect = max(maxSubRect, subRect); }          // the answer is here
```

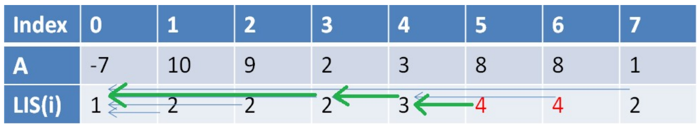
Source code: `ch3_05_UVa108.cpp/java`

From these two examples—the Max 1D and 2D Range Sum Problems—we can see that not every range problem requires a Segment Tree or a Fenwick Tree as discussed in Section 2.4.3 or 2.4.4. Static-input range-related problems are often solvable with DP techniques. It is also worth mentioning that the solution for a range problem is very natural to produce with bottom-up DP techniques as the operand is already a 1D or a 2D array. We can still write the recursive top-down solution for a range problem, but the solution is not as natural.

3. Longest Increasing Subsequence (LIS)

Given a sequence {`A[0], A[1],..., A[n-1]`}, determine its Longest Increasing Subsequence (LIS)[13]. Note that these 'subsequences' are not necessarily contiguous. Example: $n = 8$, $A = \{\underline{-7}, 10, 9, \underline{2, 3, 8}, 8, 1\}$. The length-4 LIS is {-7, 2, 3, 8}.

[13]There are other variants of this problem, including the Longest *Decreasing* Subsequence and Longest *Non Increasing/Decreasing* Subsequence. The increasing subsequences can be modeled as a Directed Acyclic Graph (DAG) and finding the LIS is equivalent to finding the Longest Paths in the DAG (see Section 4.7.1).



Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Figure 3.9: Longest Increasing Subsequence

As mentioned in Section 3.1, a naïve Complete Search that enumerates all possible subsequences to find the longest increasing one is too slow as there are $O(2^n)$ possible subsequences. Instead of trying all possible subsequences, we can consider the problem with a different approach. We can write the state of this problem with just one parameter: `i`. Let `LIS(i)` be the LIS ending at index `i`. We know that `LIS(0) = 1` as the first number in `A` is itself a subsequence. For `i` $\geq$ `1`, `LIS(i)` is slightly more complex. We need to find the index `j` such that `j` $<$ `i` and `A[j]` $<$ `A[i]` and `LIS(j)` is the largest. Once we have found this index `j`, we know that `LIS(i) = 1 + LIS(j)`. We can write this recurrence formally as:

1. `LIS(0) = 1` // the base case
2. `LIS(i) = max(LIS(j) + 1)`, $\forall$`j` $\in$ `[0..i-1]` and `A[j]` $<$ `A[i]` // the recursive case, one more than the previous best solution ending at `j` for all `j` $<$ `i`.

The answer is the largest value of `LIS(k)` $\forall$`k` $\in$ `[0..n-1]`.

Now let's see how this algorithm works (also see Figure 3.9):

- `LIS(0)` is 1, the first number in `A` = {-7}, the base case.
- `LIS(1)` is 2, as we can extend `LIS(0)` = {-7} with {10} to form {-7, 10} of length 2.
  The best `j` for `i = 1` is `j = 0`.
- `LIS(2)` is 2, as we can extend `LIS(0)` = {-7} with {9} to form {-7, 9} of length 2.
  We cannot extend `LIS(1)` = {-7, 10} with {9} as it is non increasing.
  The best `j` for `i = 2` is `j = 0`.
- `LIS(3)` is 2, as we can extend `LIS(0)` = {-7} with {2} to form {-7, 2} of length 2.
  We cannot extend `LIS(1)` = {-7, 10} with {2} as it is non-increasing.
  We also cannot extend `LIS(2)` = {-7, 9} with {2} as it is also non-increasing.
  The best `j` for `i = 3` is `j = 0`.
- `LIS(4)` is 3, as we can extend `LIS(3)` = {-7, 2} with {3} to form {-7, 2, 3}.
  This is the best choice among the possibilities.
  The best `j` for `i = 4` is `j = 3`.
- `LIS(5)` is 4, as we can extend `LIS(4)` = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}.
  This is the best choice among the possibilities.
  The best `j` for `i = 5` is `j = 4`.
- `LIS(6)` is 4, as we can extend `LIS(4)` = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}.
  This is the best choice among the possibilities.
  The best `j` for `i = 6` is `j = 4`.
- `LIS(7)` is 2, as we can extend `LIS(0)` = {-7} with {1} to form {-7, 1}.
  This is the best choice among the possibilities.
  The best `j` for `i = 7` is `j = 0`.
- The answers lie at `LIS(5)` or `LIS(6)`; both values (LIS lengths) are 4.
  See that the index `k` where `LIS(k)` is the highest can be anywhere in `[0..n-1]`.

There are clearly many overlapping sub-problems in LIS problem because to compute `LIS(i)`, we need to compute `LIS(j)` $\forall$`j` $\in$ `[0..i-1]`. However, there are only n distinct states, the indices of the LIS ending at index `i`, $\forall$`i` $\in$ `[0..n-1]`. As we need to compute each state with an $O(n)$ loop, this DP algorithm runs in $O(n^2)$.

If needed, the LIS solution(s) can be reconstructed by storing the predecessor information (the arrows in Figure 3.9) and tracing the arrows from index `k` that contain the highest value of `LIS(k)`. For example, `LIS(5)` is the optimal final state. Check Figure 3.9. We can trace the arrows as follow: `LIS(5)` $\rightarrow$ `LIS(4)` $\rightarrow$ `LIS(3)` $\rightarrow$ `LIS(0)`, so the optimal solution (read backwards) is index {0, 3, 4, 5} or {-7, 2, 3, 8}.

The LIS problem can also be solved using the *output-sensitive* $O(n \log k)$ greedy + D&C algorithm (where k is the length of the LIS) instead of $O(n^2)$ by maintaining an array that is *always sorted* and therefore amenable to binary search. Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-`i` LISs found so far. Though this definition is slightly complicated, it is easy to see that it is always ordered—`L(i-1)` will always be smaller than `L(i)` as the second-last element of any LIS (of length-i) is smaller than its last element. As such, we can binary search array `L` to determine the longest possible subsequence we can create by appending the current element `A[i]`—simply find the index of the last element in `L` that is less than `A[i]`. Using the same example, we will update array `L` step by step using this algorithm:

- Initially, at `A[0] = -7`, we have `L = {-7}`.
- We can insert `A[1] = 10` at `L[1]` so that we have a length-2 LIS, `L = {-7, 10}`.
- For `A[2] = 9`, we replace `L[1]` so that we have a 'better' length-2 LIS ending: `L = {-7, 9}`.
  This is a *greedy* strategy. By storing the LIS with smaller ending value, we maximize our ability to further extend the LIS with future values.
- For `A[3] = 2`, we replace `L[1]` to get an 'even better' length-2 LIS ending: `L = {-7, 2}`.
- We insert `A[4] = 3` at `L[2]` so that we have a longer LIS, `L = {-7, 2, 3}`.
- We insert `A[5] = 8` at `L[3]` so that we have a longer LIS, `L = {-7, 2, 3, 8}`.
- For `A[6] = 8`, nothing changes as `L[3] = 8`.
  `L = {-7, 2, 3, 8}` remains unchanged.
- For `A[7] = 1`, we improve `L[1]` so that `L = {-7, 1, 3, 8}`.
  This illustrates how the array `L` is *not* the LIS of `A`. This step is important as there can be longer subsequences *in the future* that may extend the length-2 subsequence at `L[1] = 1`. For example, try this test case: `A = {-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4}`. The length of LIS for this test case is 5.
- The answer is the largest length of the sorted array `L` at the end of the process.

Source code: `ch3_06_LIS.cpp/java`

---

4. 0-1 Knapsack (Subset Sum)

Problem[14]: Given n items, each with its own value V_i and weight W_i, $\forall i \in$ `[0..n-1]`, and a maximum knapsack size S, compute the maximum value of the items that we can carry, if we can either[15] ignore or take a particular item (hence the term 0-1 for ignore/take).

[14]This problem is also known as the Subset Sum problem. It has a similar problem description: Given a set of integers and an integer S, is there a (non-empty) subset that has a sum equal to S?

[15]There are other variants of this problem, e.g. the Fractional Knapsack problem with Greedy solution.

Example: $n = 4$, $V = \{100, 70, 50, 10\}$, $W = \{10, 4, 6, 12\}$, $S = 12$.
If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal.
If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal.
If we select item 1 and 2, we have total weight 10 and total value 120. This is the maximum.

Solution: Use these Complete Search recurrences `val(id, remW)` where `id` is the index of the current item to be considered and `remW` is the remaining weight left in the knapsack:

1. `val(id, 0) = 0` // if `remW = 0`, we cannot take anything else
2. `val(n, remW) = 0` // if `id = n`, we have considered all items
3. if `W[id]` $>$ `remW`, we have no choice but to ignore this item
`val(id, remW) = val(id + 1, remW)`
4. if `W[id]` $\leq$ `remW`, we have two choices: ignore or take this item; we take the maximum
`val(id, remW) = max(val(id + 1, remW), V[id] + val(id + 1, remW - W[id]))`

The answer can be found by calling `value(0, S)`. Note the overlapping sub-problems in this 0-1 Knapsack problem. Example: After taking item 0 and ignoring item 1-2, we arrive at state `(3, 2)`—at the third item (`id = 3`) with two units of weight left (`remW = 2`). After ignoring item 0 and taking item 1-2, we also arrive at the same state `(3, 2)`. Although there are overlapping sub-problems, there are only $O(nS)$ possible distinct states (as `id` can vary between `[0..n-1]` and `remW` can vary between `[0..S]`)! We can compute each of these states in $O(1)$, thus the overall time complexity[16] of this DP solution is $O(nS)$.

Note: The top-down version of this DP solution is often faster than the bottom-up version. This is because not all states are actually visited, and hence the critical DP states involved are actually only a (very small) subset of the entire state space. Remember: The top-down DP only visits *the required states* whereas bottom-up DP visits *all distinct states*. Both versions are provided in our source code library.

Source code: `ch3_07_UVa10130.cpp/java`

5. Coin Change (CC) - The General Version

Problem: Given a target amount V cents and a list of denominations for n coins, i.e. we have `coinValue[i]` (in cents) for coin types `i` $\in$ `[0..n-1]`, what is the minimum number of coins that we must use to represent V? Assume that we have unlimited supply of coins of any type (also see Section 3.4.1).

Example 1: $V = 10$, $n = 2$, `coinValue` $= \{1, 5\}$; We can use:
A. Ten 1 cent coins = $10 \times 1 = 10$; Total coins used = 10
B. One 5 cents coin + Five 1 cent coins = $1 \times 5 + 5 \times 1 = 10$; Total coins used = 6
C. Two 5 cents coins = $2 \times 5 = 10$; Total coins used = 2 $\rightarrow$ Optimal

We can use the Greedy algorithm if the coin denominations are suitable (see Section 3.4.1). Example 1 above is solvable with the Greedy algorithm. However, for general cases, we have to use DP. See Example 2 below:

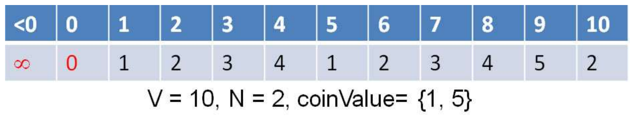
Example 2: $V = 7$, $n = 4$, `coinValue` $= \{1, 3, 4, 5\}$
The Greedy approach will produce 3 coins as its result as 5+1+1 = 7, but the optimal solution is actually 2 coins (from 4+3)!

Solution: Use these Complete Search recurrence relations for `change(value)`, where `value` is the remaining amount of cents that we need to represent in coins:

[16] If S is large such that $NS >> 1M$, this DP solution is not feasible, even with the space saving trick!

1. `change(0) = 0` // we need 0 coins to produce 0 cents
2. `change(`< 0`) =` ∞ // in practice, we can return a large positive value
3. `change(value) = 1 + min(change(value - coinValue[i]))` $\forall$`i` $\in$ `[0..n-1]`

The answer can be found in the return value of `change(V)`.



<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

Figure 3.10: Coin Change

Figure 4.2.3 shows that:
`change(0) = 0` and `change(`< 0`) =` ∞: These are the base cases.
`change(1) = 1`, from `1 + change(1-1)`, as `1 + change(1-5)` is infeasible (returns ∞).
`change(2) = 2`, from `1 + change(2-1)`, as `1 + change(2-5)` is also infeasible (returns ∞).
... same thing for `change(3)` and `change(4)`.
`change(5) = 1`, from `1 + change(5-5) = 1` coin, smaller than `1 + change(5-1) = 5` coins.
... and so on until `change(10)`.
The answer is in `change(V)`, which is `change(10) = 2` in this example.

We can see that there are a lot of overlapping sub-problems in this Coin Change problem (e.g. both `change(10)` and `change(6)` require the value of `change(5)`). However, there are only $O(V)$ possible distinct states (as `value` can vary between `[0..V]`)! As we need to try n types of coins per state, the overall time complexity of this DP solution is $O(nV)$.

A variant of this problem is to count *the number of possible (canonical) ways* to get value V cents using a list of denominations of n coins. For example 1 above, the answer is 3: {1+1+1+1+1 + 1+1+1+1+1, 5 + 1+1+1+1+1, 5 + 5}.

Solution: Use these Complete Search recurrence relation: `ways(type, value)`, where `value` is the same as above but we now have one more parameter `type` for the index of the coin type that we are currently considering. This second parameter `type` is important as this solution considers the coin types sequentially. Once we choose to ignore a certain coin type, we should not consider it again to avoid double-counting:

1. `ways(type, 0) = 1` // one way, use nothing
2. `ways(type, <0) = 0` // no way, we cannot reach negative value
3. `ways(n, value) = 0` // no way, we have considered all coin types $\in$ `[0..n-1]`
4. `ways(type, value) = ways(type + 1, value) +` // if we ignore this coin type,
`ways(type, value - coinValue[type])` // plus if we use this coin type

There are only $O(nV)$ possible distinct states. Since each state can be computed in $O(1)$, the overall time complexity[17] of this DP solution is $O(nV)$. The answer can be found by calling `ways(0, V)`. Note: If the coin values are not changed and you are given many queries with different `V`, then we can choose *not* to reset the memo table. Therefore, we run this $O(nV)$ algorithm once and just perform an $O(1)$ lookup for subsequent queries.

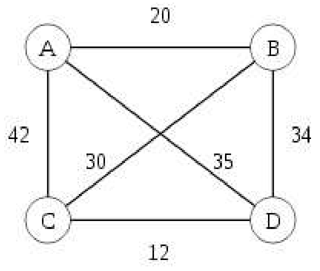
Source code (this coin change variant): `ch3_08_UVa674.cpp/java`

[17] If V is large such that $nV >> 1M$, this DP solution is not feasible even with the space saving trick!

6. Traveling Salesman Problem (TSP)

Problem: Given n cities and their pairwise distances in the form of a matrix `dist` of size $n \times n$, compute the cost of making a tour[18] that starts from any city s, goes through all the other $n - 1$ cities *exactly once*, and finally returns to the starting city s.

Example: The graph shown in Figure 3.11 has $n = 4$ cities. Therefore, we have $4! = 24$ possible tours (permutations of 4 cities). One of the minimum tours is `A-B-C-D-A` with a cost of `20+30+12+35 = 97` (notice that there can be more than one optimal solution).



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Figure 3.11: A Complete Graph

A 'brute force' TSP solution (either iterative or recursive) that tries all $O((n-1)!)$ possible tours (fixing the first city to vertex A in order to take advantage of symmetry) is only effective when n is at most 12 as $11! \approx 40M$. When $n > 12$, such brute force solutions will get a TLE in programming contests. However, if there are multiple test cases, the limit for such 'brute force' TSP solution is probably just $n = 11$.

We can utilize DP for TSP since the computation of sub-tours is clearly overlapping, e.g. the tour $A - B - C -(n - 3)$ *other cities that finally return to* A clearly overlaps the tour $A - C - B -$*the same* $(n - 3)$ *other cities that also return to* A. If we can avoid re-computing the lengths of such sub-tours, we can save a lot of computation time. However, a distinct state in TSP depends on two parameters: The last city/vertex visited `pos` and something that we may have not seen before—a *subset* of visited cities.

There are many ways to represent a set. However, since we are going to pass this set information around as a parameter of a recursive function (if using top-down DP), the representation we use must be lightweight and efficient! In Section 2.2, we have presented a viable option for this usage: The *bitmask*. If we have n cities, we use a binary integer of length n. If bit i is '1' (on), we say that item (city) i is inside the set (it has been visited) and item i is not inside the set (and has not been visited) if the bit is instead '0' (off). For example: `mask`$= 18_{10} = 10010_2$ implies that items (cities) $\{1, 4\}$ are in[19] the set (and have been visited). Recall that to check if bit i is on or off, we can use `mask & (1 << i)`. To set bit i, we can use `mask |= (1 << i)`.

Solution: Use these Complete Search recurrence relations for `tsp(pos, mask)`:

1. `tsp(pos,` $2^n - 1$`) = dist[pos][0]` // all cities have been visited, return to starting city
// Note: `mask = (1 << n) - 1` or $2^n - 1$ implies that all n bits in `mask` are on.
2. `tsp(pos, mask) = min(dist[pos][nxt] + tsp(nxt, mask | (1 << nxt)))`
// $\forall$ `nxt` $\in$ `[0..n-1]`, `nxt != pos`, and `(mask & (1 << nxt))` is '0' (turned off)
// We basically tries all possible next cities that have not been visited before at each step.

There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour. Each state can be computed in $O(n)$,

[18] Such a tour is called a Hamiltonian tour, which is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

[19] Remember that in `mask`, indices starts from 0 and are counted from the right.

thus the overall time complexity of this DP solution is $O(2^n \times n^2)$. This allows us to solve up to[20] $n \approx 16$ as $16^2 \times 2^{16} \approx 17M$. This is not a huge improvement over the brute force solution but if the programming contest problem involving TSP has input size $11 \leq n \leq 16$, then DP is the solution, not brute force. The answer can be found by calling `tsp(0, 1)`: We start from city 0 (we can start from any vertex; but the simplest choice is vertex 0) and set `mask = 1` so that city 0 is never re-visited again.

Usually, DP TSP problems in programming contests require some kind of graph preprocessing to generate the distance matrix `dist` before running the DP solution. These variants are discussed in Section 8.4.3.

DP solutions that involve a (small) set of Booleans as one of the parameters are more well known as the DP with bitmask technique. More challenging DP problems involving this technique are discussed in Section 8.3 and 9.2.

Visualization: `www.comp.nus.edu.sg/∼stevenha/visualization/rectree.html`

Source code: `ch3_09_UVa10496.cpp/java`

Exercise 3.5.2.1: The solution for the Max 2D Range Sum problem runs in $O(n^4)$. Actually, there exists an $O(n^3)$ solution that combines the DP solution for the Max Range 1D Sum problem on one dimension and uses the same idea as proposed by Kadane on the other dimension. Solve UVa 108 with an $O(n^3)$ solution!

Exercise 3.5.2.2: The solution for the `Range Minimum Query(i, j)` on 1D arrays in Section 2.4.3 uses Segment Tree. This is overkill if the given array is static and unchanged throughout all the queries. Use a DP technique to answer `RMQ(i, j)` in $O(n \log n)$ preprocessing and $O(1)$ per query.

Exercise 3.5.2.3: Solve the LIS problem using the $O(n \log k)$ solution and *also* reconstruct one of the LIS.

Exercise 3.5.2.4: Can we use an iterative Complete Search technique that tries all possible subsets of n items as discussed in Section 3.2.1 to solve the 0-1 Knapsack problem? What are the limitations, if any?

Exercise 3.5.2.5*: Suppose we add one more parameter to this classic 0-1 Knapsack problem. Let K_i denote the number of copies of item i for use in the problem. Example: $n = 2$, $V = \{100, 70\}$, $W = \{5, 4\}$, $K = \{2, 3\}$, $S = 17$ means that there are two copies of item 0 with weight 5 and value 100 and there are three copies of item 1 with weight 4 and value 70. The optimal solution for this example is to take one of item 0 and three of item 1, with a total weight of 17 and total value 310. Solve new variant of the problem assuming that $1 \leq n \leq 500$, $1 \leq S \leq 2000$, $n \leq \sum_{i=0}^{n-1} K_i \leq 40000$! Hint: Every integer can be written as a sum of powers of 2.

Exercise 3.5.2.6*: The DP TSP solution shown in this section can still be *slightly* enhanced to make it able to solve test case with $n = 17$ in contest environment. Show the required minor change to make this possible! Hint: Consider symmetry!

Exercise 3.5.2.7*: On top of the minor change asked in **Exercise 3.5.2.5***, what *other change(s)* is/are needed to have a DP TSP solution that is able to handle $n = 18$ (or even $n = 19$, but with much lesser number of test cases)?

---

[20]As programming contest problems usually require exact solutions, the DP-TSP solution presented here is already one of the best solutions. In real life, the TSP often needs to be solved for instances with thousands of cities. To solve larger problems like that, we have non-exact approaches like the ones presented in [26].

3.5.3 Non-Classical Examples

Although DP is the single most popular problem type with the highest frequency of appearance in recent programming contests, the classical DP problems in their *pure forms* usually never appear in modern ICPCs or IOIs again. We study them to understand DP, but we have to learn to solve many other non-classical DP problems (which may become classic in the near future) and develop our 'DP skills' in the process. In this subsection, we discuss two more non-classical examples, adding to the UVa 11450 - Wedding Shopping problem that we have discussed in detail earlier. We have also selected some easier non-classical DP problems as programming exercises. Once you have cleared most of these problems, you are welcome to explore the more challenging ones in the other sections in this book, e.g. Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 9.2, 9.21, etc.

1. UVa 10943 - How do you add?

Abridged problem description: Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n? Constraints: $1 \leq n, K \leq 100$. Example: For $n = 20$ and $K = 2$, there are 21 ways: $0 + 20$, $1 + 19$, $2 + 18$, $3 + 17$, $\ldots$, $20 + 0$.

Mathematically, the number of ways can be expressed as ${}^{(n+k-1)}C_{(k-1)}$ (see Section 5.4.2 about Binomial Coefficients). We will use this simple problem to re-illustrate Dynamic Programming principles that we have discussed in this section, especially the process of deriving appropriate states for a problem and deriving correct transitions from one state to another given the base case(s).

First, we have to determine the parameters of this problem to be selected to represent distinct states of this problem. There are only two parameters in this problem, n and K. Therefore, there are only 4 possible combinations:

1. If we do not choose any of them, we cannot represent a state. This option is ignored.
2. If we choose only n, then we do not know how many numbers $\leq n$ have been used.
3. If we choose only K, then we do not know the target sum n.
4. Therefore, the state of this problem should be represented by a pair (or tuple) (n, K). The order of chosen parameter(s) does not matter, i.e. the pair (K, n) is also OK.

Next, we have to determine the base case(s). It turns out that this problem is very easy when $K = 1$. Whatever n is, there is only *one way* to add exactly one number less than or equal to n to get n: Use n itself. There is no other base case for this problem.

For the general case, we have this recursive formulation which is not too difficult to derive: At state (n, K) where $K > 1$, we can split n into one number $X \in$ `[0..`n`]` and $n - X$, i.e. $n = X + (n - X)$. By doing this, we arrive at the subproblem $(n - X, K - 1)$, i.e. given a number $n - X$, how many ways can $K - 1$ numbers less than or equal to $n - X$ add up to $n - X$? We can then sum all these ways.

These ideas can be written as the following Complete Search recurrence `ways(n, K)`:

1. `ways(n, 1) = 1` // we can only use 1 number to add up to n, the number n itself
2. `ways(n, K) =` $\sum_{X=0}^{n}$ `ways(n - X, K - 1)` // sum all possible ways, recursively

This problem has overlapping sub-problems. For example, the test case $n = 1, K = 3$ has overlapping sub-problems: The state $(n = 0, K = 1)$ is reached twice (see Figure 4.39 in Section 4.7.1). However, there are only $n \times K$ possible states of (n, K). The cost of computing each state is $O(n)$. Thus, the overall time complexity is $O(n^2 \times K)$. As $1 \leq n, K \leq 100$, this is feasible. The answer can be found by calling `ways(n, K)`.

Note that this problem actually just needs the result modulo $1M$ (i.e. the last 6 digits of the answer). See Section 5.5.8 for a discussion on modulo arithmetic computation.

Source code: `ch3_10_UVa10943.cpp/java`

2. UVa 10003 - Cutting Sticks

Abridged problem statement: Given a stick of length $1 \leq l \leq 1000$ and $1 \leq n \leq 50$ cuts to be made to the stick (the cut coordinates, lying in the range `[0..`l`]`, are given). The cost of a cut is determined by the length of the stick to be cut. Your task is to find a cutting sequence so that the overall cost is minimized.

Example: $l = 100$, $n = 3$, and cut coordinates: `A` $= \{25, 50, 75\}$ (already sorted)

If we cut from left to right, then we will incur cost = 225.
1. First cut is at coordinate 25, total cost so far = 100;
2. Second cut is at coordinate 50, total cost so far = 100 + 75 = 175;
3. Third cut is at coordinate 75, final total cost = 175 + 50 = 225;

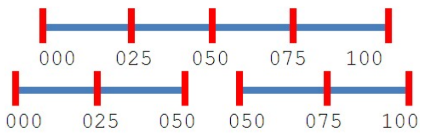


Figure 3.12: Cutting Sticks Illustration

However, the optimal answer is 200.
1. First cut is at coordinate 50, total cost so far = 100; (this cut is shown in Figure 3.12)
2. Second cut is at coordinate 25, total cost so far = 100 + 50 = 150;
3. Third cut is at coordinate 75, final total cost = 150 + 50 = 200;

How do we tackle this problem? An initial approach might be this Complete Search algorithm: Try all possible cutting points. Before that, we have to select an appropriate state definition for the problem: The (intermediate) sticks. We can describe a stick with its two endpoints: `left` and `right`. However, these two values can be very huge and this can complicate the solution later when we want to memoize their values. We can take advantage of the fact that there are only $n + 1$ smaller sticks after cutting the original stick n times. The endpoints of each smaller stick can be described by 0, the cutting point coordinates, and l. Therefore, we will add two more coordinates so that `A` $= \{0$, the original `A`, and $l\}$ so that we can denote a stick by the indices of its endpoints in `A`.

We can then use these recurrences for `cut(left, right)`, where `left`/`right` are the left/right indices of the current stick w.r.t. `A`. Originally, the stick is described by `left = 0` and `right = `n`+1`, i.e. a stick with length `[0..`l`]`:

1. `cut(i-1, i) = 0,` $\forall$`i` $\in$ `[1..`n`+1]` // if `left + 1 = right` where `left` and `right` are the indices in `A`, then we have a stick segment that does not need to be divided further.

2. `cut(left, right) = min(cut(left, i) + cut(i, right) + (A[right]-A[left]))` $\forall$`i` $\in$ `[left+1..right-1]` // try all possible cutting points and pick the best.
The cost of a cut is the length of the current stick, captured in `(A[right]-A[left])`.
The answer can be found at `cut(0, `n`+1)`.

Now let's analyze the time complexity. Initially, we have n choices for the cutting points. Once we cut at a certain cutting point, we are left with $n - 1$ further choices of the second cutting point. This repeats until we are left with zero cutting points. Trying all possible cutting points this way leads to an $O(n!)$ algorithm, which is impossible for $1 \leq n \leq 50$.

However, this problem has overlapping sub-problems. For example, in Figure 3.12 above, cutting at index 2 (cutting point = 50) produces two states: `(0, 2)` and `(2, 4)`. The same state `(2, 4)` can also be reached by cutting at index 1 (cutting point 25) and then cutting at index 2 (cutting point 50). Thus, the search space is actually not that large. There are only $(n + 2) \times (n + 2)$ possible left/right indices or $O(n^2)$ distinct states and be memoized. The time to required to compute one state is $O(n)$. Thus, the overall time complexity (of the top-down DP) is $O(n^3)$. As $n \leq 50$, this is a feasible solution.

Source code: `ch3_11_UVa10003.cpp/java`

Exercise 3.5.3.1*: Almost all of the source code shown in this section (LIS, Coin Change, TSP, and UVa 10003 - Cutting Sticks) are written in a top-down DP fashion due to the preferences of the authors of this book. Rewrite them using the bottom-up DP approach.

Exercise 3.5.3.2*: Solve the Cutting Sticks problem in $O(n^2)$. Hint: Use Knuth-Yao DP Speedup by utilizing that the recurrence satisfies Quadrangle Inequality (see [2]).

Remarks About Dynamic Programming in Programming Contests

Basic (Greedy and) DP techniques techniques are always included in popular algorithm textbooks, e.g. Introduction to Algorithms [7], Algorithm Design [38] and Algorithm [8]. In this section, we have discussed six classical DP problems and their solutions. A brief summary is shown in Table 3.4. These classical DP problems, if they are to appear in a programming contest today, will likely occur only as part of bigger and harder problems.

	1D RSQ	2D RSQ	LIS	Knapsack	CC	TSP
State	`(i)`	`(i,j)`	`(i)`	`(id,remW)`	`(v)`	`(pos,mask)`
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all $j < i$	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Table 3.4: Summary of Classical DP Problems in this Section

To help keep up with the growing difficulty and creativity required in these techniques (especially the non-classical DP), we recommend that you also read the TopCoder algorithm tutorials [30] and attempt the more recent programming contest problems.

In this book, we will revisit DP again on several occasions: Floyd Warshall's DP algorithm (Section 4.5), DP on (implicit) DAG (Section 4.7.1), String Alignment (Edit Distance), Longest Common Subsequence (LCS), other DP on String algorithms (Section 6.5), More Advanced DP (Section 8.3), and several topics on DP in Chapter 9.

In the past (1990s), a contestant who is good at DP can become a 'king of programming contests' as DP problems were usually the 'decider problems'. Now, mastering DP is a *basic* requirement! You cannot do well in programming contests without this knowledge. However, we have to keep reminding the readers of this book not to claim that they know DP if they only memorize the solutions of the classical DP problems! Try to master the art of DP problem solving: Learn to determine the states (the DP table) that can uniquely and efficiently represent sub-problems and also how to fill up that table, either via top-down recursion or bottom-up iteration.

There is no better way to master these problem solving paradigms than solving real programming problems! Here, we list several examples. Once you are familiar with the examples shown in this section, study the newer DP problems that have begun to appear in recent programming contests.

Programming Exercises solvable using Dynamic Programming:

- Max 1D Range Sum
  1. UVa 00507 - Jill Rides Again (standard problem)
  2. **UVa 00787 - Maximum Sub ... *** (max 1D range *product*, be careful with 0, use Java BigInteger, see Section 5.3)
  3. **UVa 10684 - The Jackpot *** (standard problem; easily solvable with the given sample source code)
  4. ***UVa 10755 - Garbage Heap **** (combination of max 2D range sum in two of the three dimensions—see below—and max 1D range sum using Kadane's algorithm on the third dimension)
     See more examples in Section 8.4.
- Max 2D Range Sum
  1. **UVa 00108 - Maximum Sum *** (discussed in this section with sample source code)
  2. UVa 00836 - Largest Submatrix (convert '0' to -INF)
  3. UVa 00983 - Localized Summing for ... (max 2D range sum, get submatrix)
  4. UVa 10074 - Take the Land (standard problem)
  5. UVa 10667 - Largest Block (standard problem)
  6. **UVa 10827 - Maximum Sum on ... *** (copy $n \times n$ matrix into $n \times 2n$ matrix; then this problem becomes a standard problem again)
  7. ***UVa 11951 - Area **** (use long long; max 2D range sum; prune the search space whenever possible)
- Longest Increasing Subsequence (LIS)
  1. UVa 00111 - History Grading (be careful of the ranking system)
  2. UVa 00231 - Testing the Catcher (straight-forward)
  3. UVa 00437 - The Tower of Babylon (can be modeled as LIS)
  4. **UVa 00481 - What Goes Up? *** (use $O(n \log k)$ LIS; print solution; see our sample source code)
  5. UVa 00497 - Strategic Defense Initiative (solution must be printed)
  6. UVa 01196 - Tiling Up Blocks (LA 2815, Kaohsiung03; sort all the blocks in increasing `L[i]`, then we get the classical LIS problem)
  7. UVa 10131 - Is Bigger Smarter? (sort elephants based on decreasing IQ; LIS on increasing weight)
  8. UVa 10534 - Wavio Sequence (must use $O(n \log k)$ LIS twice)
  9. *UVa 11368 - Nested Dolls* (sort in one dimension, LIS in the other)
  10. **UVa 11456 - Trainsorting *** (max(LIS(i) + LDS(i) - 1), $\forall i \in [0 \ldots n\text{-}1]$)
  11. **UVa 11790 - Murcia's Skyline *** (combination of LIS+LDS, weighted)

- 0-1 Knapsack (Subset Sum)
  1. UVa 00562 - Dividing Coins (use a one dimensional table)
  2. UVa 00990 - Diving For Gold (print the solution)
  3. UVa 01213 - Sum of Different Primes (LA 3619, Yokohama06, extension of 0-1 Knapsack, use three parameters: (id, remN, remK) on top of (id, remN))
  4. UVa 10130 - SuperSale (discussed in this section with sample source code)
  5. UVa 10261 - Ferry Loading (s: current car, left, right)
  6. **UVa 10616 - Divisible Group Sum *** (input can be -ve, use long long)
  7. UVa 10664 - Luggage (Subset Sum)
  8. **UVa 10819 - Trouble of 13-Dots *** (0-1 knapsack with 'credit card' twist!)
  9. *UVa 11003 - Boxes* (try all max weight from 0 to $max(weight[i]+capacity[i])$, $\forall i \in [0..n\text{-}1]$; if a max weight is known, how many boxes can be stacked?)
  10. UVa 11341 - Term Strategy (s: id, h_learned, h_left; t: learn module 'id' by 1 hour or skip)
  11. ***UVa 11566 - Let's Yum Cha **** (English reading problem, actually just a knapsack variant: double each dim sum and add one parameter to check if we have bought too many dishes)
  12. UVa 11658 - Best Coalition (s: id, share; t: form/ignore coalition with id)
- Coin Change (CC)
  1. UVa 00147 - Dollars (similar to UVa 357 and UVa 674)
  2. UVa 00166 - Making Change (two coin change variants in one problem)
  3. **UVa 00357 - Let Me Count The Ways *** (similar to UVa 147/674)
  4. UVa 00674 - Coin Change (discussed in this section with sample source code)
  5. **UVa 10306 - e-Coins *** (variant: each coin has two components)
  6. UVa 10313 - Pay the Price (modified coin change + DP 1D range sum)
  7. UVa 11137 - Ingenuous Cubrency (use long long)
  8. **UVa 11517 - Exact Change *** (a variation to the coin change problem)
- Traveling Salesman Problem (TSP)
  1. **UVa 00216 - Getting in Line *** (TSP, still solvable with backtracking)
  2. **UVa 10496 - Collecting Beepers *** (discussed in this section with sample source code; actually, since $n \leq 11$, this problem is still solvable with recursive backtracking and sufficient pruning)
  3. **UVa 11284 - Shopping Trip *** (requires shortest paths pre-processing; TSP variant where we can go home early; we just need to tweak the DP TSP recurrence a bit: at each state, we have one more option: go home early)

  See more examples in Section 8.4.3 and Section 9.2.
- Non Classical (The Easier Ones)
  1. UVa 00116 - Unidirectional TSP (similar to UVa 10337)
  2. *UVa 00196 - Spreadsheet* (notice that the dependencies of cells are acyclic; we can therefore memoize the direct (or indirect) value of each cell)
  3. *UVa 01261 - String Popping* (LA 4844, Daejeon10, a simple backtracking problem; but we use a `set<string>` to prevent the same state (a substring) from being checked twice)
  4. UVa 10003 - Cutting Sticks (discussed in details in this section with sample source code)
  5. UVa 10036 - Divisibility (must use offset technique as value can be negative)

6. *UVa 10086 - Test the Rods* (s: idx, rem1, rem2; which site that we are now, up to 30 sites; remaining rods to be tested at NCPC; and remaining rods to be tested at BCEW; t: for each site, we split the rods, x rods to be tested at NCPC and $m[i] - x$ rods to be tested at BCEW; print the solution)
7. **UVa 10337 - Flight Planner *** (DP; shortest paths on DAG)
8. UVa 10400 - Game Show Math (backtracking with clever pruning is sufficient)
9. *UVa 10446 - The Marriage Interview* (edit the given recursive function a bit, add memoization)
10. UVa 10465 - Homer Simpson (one dimensional DP table)
11. *UVa 10520 - Determine it* (just write the given formula as a top-down DP with memoization)
12. *UVa 10688 - The Poor Giant* (note that the sample in the problem description is a bit wrong, it should be: 1+(1+3)+(1+3)+(1+3) = 1+4+4+4 = 13, beating 14; otherwise a simple DP)
13. **UVa 10721 - Bar Codes *** (s: n, k; t: try all from 1 to m)
14. UVa 10910 - Mark's Distribution (two dimensional DP table)
15. UVa 10912 - Simple Minded Hashing (s: len, last, sum; t: try next char)
16. **UVa 10943 - How do you add? *** (discussed in this section with sample source code; s: n, k; t: try all the possible splitting points; alternative solution is to use the closed form mathematical formula: $C(n + k - 1, k - 1)$ which also needs DP, see Section 5.4)
17. *UVa 10980 - Lowest Price in Town* (simple)
18. *UVa 11026 - A Grouping Problem* (DP, similar idea with binomial theorem in Section 5.4)
19. UVa 11407 - Squares (can be memoized)
20. UVa 11420 - Chest of Drawers (s: prev, id, numlck; lock/unlock this chest)
21. UVa 11450 - Wedding Shopping (discussed in details in this section with sample source code)
22. UVa 11703 - sqrt log sin (can be memoized)

- Other Classical DP Problems in this Book
  1. Floyd Warshall's for All-Pairs Shortest Paths problem (see Section 4.5)
  2. String Alignment (Edit Distance) (see Section 6.5)
  3. Longest Common Subsequence (see Section 6.5)
  4. Matrix Chain Multiplication (see Section 9.20)
  5. Max (Weighted) Independent Set (on tree, see Section 9.22)
- Also see Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 8.4 and parts of Chapter 9 for *more* programming exercises related to Dynamic Programming.